Transformative and troublesome? Students' and professional programmers' perspectives on difficult concepts in programming

Article (Published Version)

Transformative and Troublesome? Students' and professional programmers' perspectives on difficult concepts in programming

LUCY YEOMANS, School of Education, Communication & Society, King's College London
STEFFEN ZSCHALER, Department of Informatics, King's College London
KELLY COATE, School of Education and Social Work, University of Sussex

Programming skills are an increasingly desirable asset across disciplines; however, learning to program continues to be difficult for many students. To improve pedagogy, we need to better understand the concepts that students find difficult and which have the biggest impact on their learning. Threshold-concept theory provides a potential lens on student learning, focusing on concepts that are troublesome and transformative. However, there is still a lack of consensus as to what the most relevant threshold concepts in programming are. The challenges involved are related to concept granularity and to evidencing some of the properties expected of threshold concepts. In this paper, we report on a qualitative study aiming to address some of these concerns. The study involved focus groups with undergraduate students of different year groups as well as professional software developers so as to gain insights into how perspectives on concepts change over time. Four concepts emerged from the data, where the majority of participants agreed on their troublesome nature—including abstract classes and data structures. Some of these concepts are considered transformative, too, but the evidence base is weaker. However, even though these concepts may not be considered transformative in the 'big' sense of threshold concept theory, we argue the 'soft' transformative effect of such concepts means they can provide important guidance for pedagogy and the design of programming courses. Further analysis of the data identified additional concepts that may hinder rather than help the learning of these threshold concepts, which we have called 'accidental complexities'. We conclude the paper with a critique of the use of threshold concepts as a lens for studying students' learning of programming.

CCS Concepts: •**Social and professional topics** → **CS1; Software engineering education;** •**Software and its engineering** → *Object oriented development;*

Additional Key Words and Phrases: threshold concepts; learning programming; focus groups; computer science curriculum; accidental complexities

ACM Reference format:
Lucy Yeomans, Steffen Zschaler, and Kelly Coate. 2018. Transformative and Troublesome? Students' and professional programmers' perspectives on difficult concepts in programming. *ACM Trans. Comput. Educ.* 1, 1, Article 1 (January 2018), 27 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Programming is an essential skill for any computer science student. Indeed, many would argue that most STEM (Science, Technology, Engineering and Maths) subjects would expect their students to have some understanding and capability in programming as part of their course [31]. Programming

This work was supported by a King's College Teaching Fund grant.
 1946-6226/2018/1-ART1 $15.00
DOI: 0000001.0000001

skills are seen to underpin many other subjects, teaching the importance of accuracy and attention to detail as well as developing the capacity for problem-solving and creating solutions to real-world problems [16, 18, 48]. The recent introduction of compulsory computer science education in many countries has put programming high on the agenda, with pupils as young as 5 years old in the United Kingdom expected to learn how to create and debug simple programs [5].

Nevertheless, learning to program is generally acknowledged to be very difficult: students are expected to have the correct abstract understanding of a concept and be able to implement it in a concrete manner using appropriate strategies [25, 40], requiring a substantial amount of hands-on programming experience. A significant body of literature has been devoted to exploring the subject-specific difficulties beginner programmers face. Eckerdal *et al.* [14] found novice programmers were less likely to grasp the interrelation between a program's parts and the program as a whole. Sorva [47] suggests that, unlike in many other disciplines, programming students are less likely to be encouraged to subjectively interpret and apply their learning, with many concepts being precisely defined and implemented. Such strict parameters lead to undesirable and unproductive programming outcomes. Other studies have argued novice programmers typically have a superficial understanding of programming that is context specific; therefore, they struggle with knowledge transfer [25, 30]. Such factors were apparent in a range of studies on computer-science students' programming competencies, which found many students performing well below expectations [28, 31]. This issue is compounded by an apparent lack of mechanisms available to educators for determining the level of programming skill acquisition students should be attaining [32]. Furthermore, the heterogeneity of student cohorts regarding their experience of programming makes differentiation extremely challenging; this is often attributed to be one of the major factors contributing to high drop-out rates from university courses [21, 25, 37].

We experienced similar challenges in our own teaching of programming at King's. In response, we made several changes to our teaching. The programming module focused on in this research was traditionally taught through a conventional lecture format followed by lab sessions in which individual students worked through programming tasks. We revised the pedagogical approach taken in the module, including lecture sessions with more interactive learning opportunities and the introduction of pair rather than individual programming in the labs. Despite some improvement, the progression rate of students remained a cause for concern, prompting one of the authors, a module leader, to collaborate with an higher-education researcher and a science-education researcher to undertake a project investigating the learning experience of student programmers.

The notion of 'threshold concepts' [34] can provide a useful lens for identifying critical points in students' learning journeys. We set out to use this lens to explore what students find challenging when learning to programme and how we might adjust our teaching to help them overcome these challenges. While there is already a substantial body of work on threshold concepts in programming (and in computing more widely) [42], there appears to be a lack of consensus on what these concepts are. More empirical research is required—in particular including triangulation with participants other than (the traditionally chosen) teaching staff and students. Our study sampled undergraduate students from different year groups as well as professional software developers, seeking to identify candidate threshold concepts based on recurrence of concepts between these groups. In identifying concepts, we particularly looked out for markers of difficulty (aligned with the definition of threshold concepts as 'troublesome') and of change of perspective (as per the definition of threshold concepts as 'transformative').

Our primary research question was: "What are threshold concepts for learners of (object-oriented) programming?" As we progressed through our study, we added a secondary research question:

"Are threshold concepts a suitable, and pedagogically useful, concept in understanding learning of (object-oriented) programming?"

Here, we report on the outcomes of this research, beginning with a discussion of theoretical approaches in Sect. 2 and our methodology in Sect. 3. The findings are reported in Sect. 4 and then discussed in Sect. 5. Finally, we draw conclusions about the implications of the study for programming education and offer suggestions for future research directions.

2 THEORETICAL APPROACHES TO STUDENT LEARNING IN PROGRAMMING

A number of different theoretical lenses have been used to explore challenges faced by student programmers. A great deal of literature exists on misconceptions in introductory programming [22]. For example, Clancy [7] argues that misconceptions and mistaken attitudes complicate learning and suggests a range of sources in programming, including linguistic issues, confusion with mathematical notation, and transferring new knowledge into inappropriate settings. However, while misconceptions indicate a 'failure to learn' [13], it is argued a misconception lens is limited at uncovering the 'conceptual jewels' of any given discipline. Holland et al. [19] suggest that misconceptions in programming do not always occur because a concept is, in itself, difficult. Instead, they may occur, for example, because students conflate everyday understandings of a term with the term's technical meaning. Thus, the presence of a misconception does not, by itself, indicate a core concept [13]. An alternative approach which does focus on the 'conceptual jewels' of programming has been the use of 'fundamental ideas'. Fundamental ideas are broad, lasting, and of perhaps universal significance [46]. They connect ideas within and beyond a subject field, are widely applicable—across disciplines, across time, across levels of expertise—and connect the academic with everyday life. However, fundamental ideas are not likely to be transformative, and are not necessarily challenging to learn [51].

Our goal for this research was to identify the concepts that students struggled with when learning programming and where improving student understanding would have the biggest impact. Neither the lens of misconceptions nor that of fundamental ideas seemed to fit this goal, while the troublesome and transformative elements of threshold-concept theory provided a better fit. We will discuss threshold-concept theory and its application in programming education in the following sub-sections.

2.1 Threshold Concepts

The notion of 'threshold concepts' has been used in the wider education literature to identify concepts that are central to students' mastering of a particular subject area. Threshold concepts can be defined as core 'gateway' concepts which unlock new, previously inaccessible knowledge and may be in themselves particularly difficult to overcome [34]. Meyer and Land were the first to introduce the notion of threshold concepts and in 2005 described them as theoretical summits that, once reached, signify either a leap forward in an individual's understanding, a clarity of a concept's complexity and how it connects to other ideas, or a point at which a significant idea becomes embedded within someone's knowledge in such a way that it would be hard to undo. They subsequently simplified this definition into four characteristics of threshold concepts: *Troublesome, Transformative, Integrated* and *Irreversible.*

Since Meyer and Land's initial proposal the uptake of a threshold concept approach to teaching and learning in higher education has been enthusiastic. Nevertheless, the definition and process of identification of threshold concepts remains subjective and contested [2, 42]. Firstly, there is a lack of consensus as to how many of the four characteristics are required to make a concept a 'threshold' rather than simply a 'core' concept [2]. Occasionally, threshold concepts are also required to be

boundary markers, marking the limits of a subject area [13]. Furthermore, Davies [10] concedes that threshold concepts may be additionally difficult to identify because within any given discipline they may be 'taken for granted' and as such will not be made explicit.

Later development of threshold-concept theory proposed that threshold concepts are rarely grasped during one 'eureka' moment. Instead, it was suggested that they require a transitional period—from one state of being and knowing to another [41]. Such transitional periods, or 'liminal spaces', are where students are most likely to get stuck [41]. Studies have also suggested that more nuanced aspects of the *troublesome, transformative, integrated* and *irreversible* should be considered when understanding the nature of the problems faced by learners. For example, students may use 'mimicry' as a way of grappling with difficult, troublesome concepts, reproducing what they have been shown without concrete understanding. Such mimicry can be viewed, however, as a useful step towards more complete understanding of a concept while in the liminal space [34, 41].

2.2 Threshold Concepts in Programming Education

The challenging nature of learning to program has made threshold-concept theory an attractive approach for educators in the field. Sanders and McCartney's recent survey [42] provides a good overview of research undertaken in this area, including a list of some concepts identified by various authors, and highlights a number of open challenges. Previous studies on threshold concepts in programming have had limited success in identifying suitable candidates. This was either due to lack of consensus, resulting in too many concepts being nominated [44]; or because the concepts nominated covered too large a theoretical area. Object-oriented programming (OOP), for example, has been suggested several times in the literature as a prospective threshold concept in programming (*e.g.,* [4, 12, 35, 44]), but it has also been criticized as being far too large an area to be of significant use [41]. Furthermore, other studies have sampled their participants from a university context only (students and lecturers) [42]. So far there have been no studies which incorporate the views of industry experts. As threshold concepts may take a long time to bed in, there is, thus, a gap in our current understanding, as has also been acknowledged in the literature [2, 41].

The notion of mimicry has been identified as particularly relevant for programming. This is a common approach to studying used by students managing difficult course requirements, but is, on its own, less effective than deeper or strategic approaches [20]. Eckerdal *et al.* identified a framework incorporating mimicry to be used for pinpointing when and where a student is in a liminal space while learning to program [15]. The framework was based on different sorts of conceptual understanding: abstract / theoretical understanding of a concept; concrete understanding of the concept evidenced through practical programming; the ability to go from abstract to concrete understanding; understanding why the concept is used and taught and understanding the application of the concept in new situations.

Eckerdal *et al.*'s study also discussed the emotional response associated with liminal spaces and programming in particular. They argue that the strong emotional response from students facing difficulties when learning to program is often ignored in the literature, while they should be recognised as 'normal and desirable' by educators [15]. Furthermore, an emotional reaction to a concept was regarded by Rountree and Rountree to be an indicator of its potential to be a threshold concept, as both frustration and elation could show where a student is within liminal space [41].

An additional dimension of the *transformative* characteristic proposed by Eckerdal *et al.* was that once students master the threshold concepts of their field they should have become familiar and comfortable with the central ideas of their field. As such, they can be said to have acquired a new identity, that of an 'insider' within a discipline rather than a student practising their subject [15, 41].

As programming skills have been identified as so difficult to master, we would expect this 'feeling like an expert' to be particularly pronounced when students master threshold concepts in programming.

2.3 Theoretical Framework

The theoretical framework used in the study focussed on troublesome and transformative knowledge, the first two characteristics of threshold concepts suggested by Meyer and Land [34]. Sanders and McCartney [42] argue that the quality of transformation is the only quality essential in threshold concepts. We would add that the dimension of difficulty provides useful indicators to help identify potential conceptual candidates through its association with the concept of liminal space [4]. For these reasons, we chose to prioritize the troublesome and transformative characteristics of threshold concepts in our study.

The framework also incorporated Eckerdal *et al.*'s notions of conceptual understanding to identify instances associated with troublesome knowledge, where only *partial understanding* had taken place and perhaps even unbeknownst to the participants themselves [15]. Partial understanding could be evidenced through abstract understanding of concepts without the ability to use them in a concrete manner and *vice versa*. We also looked for evidence of 'mimicry', or the copying of the work of others or examples of work found elsewhere. Eckerdal *et al.*'s work also suggests that 'naïve' knowledge may also be evidence of partial understanding, where students on reflection realize that concepts they thought they had mastered were much harder than originally thought, and perhaps still out of reach. We made further use of Eckerdal *et al.*'s approach through providing multiple facets of the transformative characteristic, including the 'feeling like a programmer' or 'insider' and looked for additional clues as to a student being in liminal space such as evidence of an emotional response.

Table 1 outlines the elements of threshold concept theory drawn upon as part of the study, both in developing the methods used and the subsequent analysis to identify threshold concepts in programming.

3 METHODOLOGY

The study took place over the academic year 2015-2016 at King's College London, where all three authors were based. Programming is introduced to students in the Department of Informatics through a large-class (approx. 350 students per year) first-year undergraduate module as part of a three-year BSc programme. As part of the overall programme, students develop their programming skills through a range of group and individual projects and assignments. As in other higher-education contexts [37], students' previous experience with programming varies widely. In a base-line survey[1], approximately 52% of respondents self-identified as 'beginners' with 'no or very little programming experience'. At the same time, approximately 30% of respondents claimed they had worked on smaller or larger object-oriented programs.[2]

The methodology was influenced by a desire to improve the learning experience for students on the module. The following sub-sections discuss the participants sampled, the methods chosen and the study's approach to data analysis to achieve this aim, as well as the strengths and limitations of these decisions and processes.

3.1 Participants

Data were collected from three sample populations: first-year undergraduate students ($N = 9$), third-year undergraduate students ($N = 3$) and professional software developers with a range of

[1]Based on the survey developed by Pedroni *et al.* [37]

[2]The boundary between small and large was set at approximately 100 classes

Characteristic	Description
Troublesome knowledge	Indications of participants getting 'stuck' or ideas which have taken time to overcome
Transformative knowledge	Evidence of gaining new perspectives or transformation of view on broad conceptual area
Integrated knowledge	Concepts which lead to seeing how existing knowledge is linked together
Irreversible knowledge	Suggestions that a concept or idea is now permanently embedded
Liminal space	Suggestions that participants are in the transitional period between beginning to learn a new concept and being proficient with it—for example, an emotional response to the learning process, either positive or negative
Partial understanding	Evidence of partial understanding having taken place such as: mimicry of understanding; abstract understanding without concrete understanding; being unable to apply a concept in a new or different context; concrete understanding without abstract understanding; naïve version of knowledge; (not) understanding the rationale for learning a concept; contextualized learning
Feeling like an 'insider'	Participants talk of feeling like a programmer; evidence of students' use of new language; indications of subjectivity, where participants have repositioned themselves when talking about ideas related to the field

Table 1. Threshold concept theoretical framework

programming experience ($N = 5$). The rationale was to capture a sense of which concepts were identified as particularly challenging at the beginning and end of the undergraduate experience, as well as incorporating the perspectives of practitioners far more established in the field. The emphasis of subject expertise within threshold concept theory [43] suggests that experts are required in a study using the threshold-concept lens. The view of experts was drawn upon because their familiarity with the subject knowledge meant they could know where in a body of knowledge to 'look' [44]. The decision was made not to include the perspectives of faculty staff as, while they could be considered experts in their field, their perspective would likely be as expert-teachers rather than from their own personal experiences of learning to program. Professional software-developers, meanwhile, are not faced with the same potential limitation and, furthermore, bring in expertise from fields beyond the academic environment [1, 41]. Practitioners, in particular, have not been involved in threshold-concept research so far [42]; one of our aims was to address this gap.

Nevertheless, sampling from professional software developers still has potential drawbacks. For example, professional pride may prevent them from acknowledging finding certain concepts difficult and there is the added danger of hindsight bias. The transformative nature of threshold concepts means that experts may be in a position where, having crossed the threshold a long time ago, the knowledge can become tacit and 'taken for granted' [10, 34] Concerns regarding hindsight bias for both the practitioners and, to a certain extent, the third-year undergraduate students can be mitigated somewhat by the involvement of the first-year undergraduates in the study. As will be discussed further below, the study aimed to find commonality of responses from the three

different sample populations in order to validate our findings. To counter concerns of 'professional pride' as mentioned above, the main focus of the discussions were practitioners' experiences of learning to program, rather than the concepts they continued to struggle with. However, as will be explored more in the findings, many of the participants from this sample volunteered examples of the ongoing challenges of programming.

The undergraduate students (1 female and 11 male) were of traditional university age, came from a range of backgrounds, including home and international students, and had a range of experience with programming. The software developers were male and aged between approximately 20 and 60 years old. They had followed various educational and professional paths to become professional programmers and had differing amounts of experience.

A variety of methods were used to recruit participants:

- *First-year undergraduate students.* To familiarise students with the researcher and the research, the first author—who later undertook the focus group sessions—was introduced to the students at the start of one of their first-year programming lectures by the second author—who was teaching that module. Later, the first author observed a number of lab sessions as part of the first phase of data collection and recruitment for the focus-group sessions. In addition, posters and emails (sent by Departmental administrators) were used to recruit participants.
- *Third-year undergraduate students.* The first author directly recruited third-year students through conversations in the computing lab, where they were working on their coursework.
- *Professional software developers.* This group of participants was recruited from London Java Community through emails sent by the community coordinator to members. We did not establish whether any of these participants had previously studied at King's. Some professional developers did in fact not have an undergraduate degree in programming at all. They were, however, not specifically recruited because of their education path.

Participants were offered no remuneration for their participation. They were provided with detailed information about the project and all participants gave written consent. Throughout the research, it was clear to all (potential) participants that participation was voluntary and would, in particular, not have any impact on their learning or assessment outcomes.

3.2 Methods

The study collected a range of qualitative data, shaped by the theoretical framework based on threshold-concept theory (*cf.* Sect. 2.3). The instruments were designed with a focus on drawing out examples of troublesome and transformative concepts, with questions related to aspects of programming that participants have / had struggled with as well as which had potentially provided them with a paradigm-shift in understanding of the subject matter. The study used two different approaches to elicit nominations for threshold concepts from each participant, to help explore tacit knowledge and enable the identification of threshold concepts where participants might have only partial awareness. These approaches are described in detail below.

All data collection was undertaken by the first author, an educational researcher without programming background who had no teaching relationship with student participants. Throughout the study, the first author explicitly told participants that she had limited previous knowledge in computer science. This was done to ensure participants (in particular first-year students) felt free to discuss whichever programming concepts they found to be pertinent without fear of criticism.

There were two phases of data collection:

(1) The first phase was comprised of observations and unstructured interviews with first-year undergraduate students participating in pair-programming labs over a three-month period.

Students were asked questions regarding their experiences of programming and which aspects of the course they enjoyed and which they found difficult. These observations and unstructured interviews took place during the lab sessions. The first author observed and informally questioned students completing lab exercises. Observations and unstructured interviews were recorded in field notes of the first author. While these data were not extensively used in the final analysis, they were drawn on to inform the questions asked during the next data collection phase.

(2) The second phase encompassed five focus groups with a range of participants[3], conducted to investigate potential threshold concepts in a more directed manner. Focus groups were chosen as the main data collection method as they can be used to gauge to what extent views are shared amongst a group of people [11]. This seemed the most appropriate approach to explore the problem the study was attempting to address, as discussed further below. Furthermore, focus groups have been argued to facilitate the democratisation of the research process by giving participants greater ownership over the discussions and redirect power from the interviewer [23]. While this was a desirable outcome for our study, it brought with it the potential of participants deviating from the subject matter, talking over one another (thereby making transcription difficult), or having one or more participants dominating the group. The study mediated these issues by using an experienced facilitator to manage the discussions. On a practical note, focus groups also helped with recruitment as, in our experience, undergraduate students are less likely to attend individual interviews.

Focus groups were moderated by the first author and consisted of two parts:

(a) In the first part of the focus group, the first author asked participants open questions regarding aspects of programming they found enjoyable and aspects they found challenging. Students were also asked to self-assess their level of programming experience. This was done to give participants a general sense of the nature of the study and to provide them an opportunity to offer responses which were not influenced by the prompts used in the second half of the focus group as detailed below. Indicative focus-group prompts can be found in Appendix A.

(b) The introductory questions were then followed by a structured activity where participants were asked, as a group, to physically organize a list of concepts covered in our curriculum in order of difficulty (see Fig. 1). A list of concepts provided on these cards can be found in Appendix B. Participants were also provided empty cards to add to the list, which were used twice during the study (adding polymorphism, and functional programming). The final arrangement was then used as a prompt in further questioning related to potential threshold concepts as discussed above.

At no point were participants' descriptions of concepts challenged or verified by the facilitator. Focus group sessions were audio-recorded and, subsequently, transcribed by an external agency.

Three focus groups were held with undergraduate students in their first year, who were asked to reflect on their experiences of the first two terms. A further focus group took place with undergraduate students in their third year, where they were asked to consider both their experiences in their first year and their current views, and a final focus group was held with professional programmers who came from a range of training backgrounds with varying levels of experience. The industry professionals were asked to recount their experiences of learning programming and invited to discuss what they continued to encounter as problematic. The aim in this approach was to

[3] 3 first-year groups with 2, 1 (because of scheduling constraints), and 6 participants, respectively; 1 third-year group with 3 students; and 1 group of professional developers with 5 participants

Fig. 1. Card sorting activity

find some commonality between the concepts proposed from the three different sample populations and then put the resulting nominations under further scrutiny as candidates for threshold concepts.

3.3 Analysis

All focus groups were digitally audio-recorded and transcribed, while photos were taken of the sorting activity to record the final arrangement of concepts as decided by participants. Data were analysed using a mostly deductive approach, using codes developed through the threshold-concept framework as outlined above. Transcripts were initially coded and sorted by the first author, using the NVivo software package, into specific programming concepts, and participants' references to topic difficulty, transformation of understanding and reported examples of partial understanding in its different forms. Our interest in exploring liminal space meant attention was particularly given to data related to difficulties participants had faced during their training and the emotional responses which may have accompanied these issues. This approach led to the addition of supplementary codes, details of which are discussed below under the heading of 'additional findings'. Once anonymized, the first and second author undertook a further analysis round to ensure the initial findings were valid and no concepts were overlooked by the educational researcher. Through this second analysis, additional concepts were identified which appeared to be only partially understood by the participants, even when this partial understanding was not explicitly identified by participants. These instances were coded separately for clarity. Programming concepts which were mentioned in three or more of the five focus groups, with one of them being the practitioner group, were analyzed for evidence of being both troublesome and transformative. Concepts which

Programming Concept	Dimension of Threshold Concept Theory	Data Example
Classes and inheritance	*Troublesome* and *transformative* by practitioners, *troublesome* and *transformative* by students	"Classes and inheritance for me…yes, getting your head around that was quite hard but then once you, it becomes quite a vital part of the programming once you get your head around it" *year 1 student participant*
User Interface Architectures	*Transformative* by practitioners, *troublesome* and *transformative* by students	"The MVC [Model View Controller] for me it's, I don't really get it yet like how to actually do it but, yes, it actually makes the programme neater and stuff, yes." *year 1 student participant*
Data structures	*Troublesome* by practitioners, *troublesome* and *transformative* by students	"(Data Structures) weren't hard to understand, but we don't, we all understand, we should understand it a little bit better than what we currently do. And definitely the first year we felt that way" *year 3 student participant*
Abstract classes	*Troublesome* and *transformative* by practitioners, *troublesome* by students	"And abstract classes for me anyway, I don't know about anyone else…Yes, I mean I understand it I just don't know how like I just I've never seen the point in using it" *year 1 student participant*

Table 2. Identified threshold concepts

were subsequently identified as having just one of the characteristics were discarded. The remaining concepts are outlined below.

4 FINDINGS

4.1 Candidate Threshold Concepts

Following the analysis, four candidates for threshold concepts in programming emerged. A summary can be found in Table 2.[4] Below, we discuss each concept in more detail. In these discussions, we begin by giving a brief description of the concept before discussing how this concept was interpreted by our participants and what led to it being included as a candidate threshold concept. It should be noted that the initial descriptions we provide below are our own definitions and that they were not provided to or elicited from the participants.

4.1.1 Classes and inheritance. Classes are a fundamental notion in object-oriented programming, encapsulating state and behaviour of the objects being described (sometimes referred to as "simulated") by the program. Classes, then, can be used to instantiate new objects (*i.e.,* they act as a blueprint for new objects), but they can also serve to type (or "classify") objects. Inheritance defines relationships between classes, enabling some classes to obtain copies of (or "inherit") features of other classes. When a class `A` inherits from a class `B` (viewing classes as blueprints), we can also say that `B` generalises `A` (from a type perspective).

[4]Appendix C provides some more detailed statistics about the occurrence of individual concepts across the whole data set.

'Classes and inheritance' was a relatively straightforward nomination for a threshold concept. It was mentioned in all five of the focus groups and both students and practitioners described it in terms that could be identified as troublesome and transformative. One of the first-year undergraduate focus groups specifically spoke of classes and inheritance as *transformative*:

> *Both, I didn't know you could inherit things from other classes originally so it improved my understanding and helped with my program*
>
> *(Year 1 student)*

Some students in the third year particularly singled out 'classes and inheritance' as a concept for which they needed additional support:

> **Respondent 1** *Do you remember the abstract class and inheritance?*
> **Respondent 2** *That's hard though.*
> **Respondent 3** *That was hard.*

4.1.2 User Interface Architectures. User interfaces are where a software system interacts with its users and *vice versa.* Modern user interfaces, particularly graphical user interfaces (GUIs), often make use of a event-driven style of programming, where the software waits to receive events from the user (*e.g.,* the user types a character, or moves the mouse cursor to a new location) and then handles these events and responds by, ultimately, modifying features of the GUI in an appropriate way. This usually involves a substantial number of objects inter-operating in a non-trivial manner. The structure of these object interactions is sometimes referred to as the user-interface architecture.

Some of the first-year undergraduates implied they found 'user interface architectures' *troublesome,* with one mentioning a partial understanding characterized by a theoretical but not a concrete grasp of the idea:

> *I sort of understand Regular Expressions and I can implement them whereas stuff like User Interface Architectures. Like, 'comfortable' is different, because it's the implementation and the theory behind it. Like Layout Management Concepts are quite easy because there's like a visual guide to how they're laid out, but actually making them bend to your will, per se, is quite fidgety.*
>
> *(Year 1 student)*

First year students in another focus group described user interface architectures in language that could be identified as indicating a *transformative* and *troublesome* concept. However, the third-year students didn't allude to the concept at all. The practitioners suggested the idea of user interface architectures as a gateway concept, as the ability to master it led to them seeing how programming can be used in a real-world context rather than simply solving an exercise in the classroom:

> *Well I guess I think the thing that makes me feel real about it is the real-world side of things, which is the user interface architecture or the real-world entities... Just being able to build something which* does *something.*
>
> *(Practitioner, emphasis added)*

4.1.3 Data structures. Any computation requires storing data in memory. The choice of data structure (*i.e.,* the way in which data are stored and cross-referenced) can substantially influence the efficiency of the algorithms processing the data. Data structures usually taught at university level include various types of list structures, queues, stacks, and maps.

While some concepts were referred to in language indicating troublesome or transformative nature in a relatively fleeting manner, data structures were discussed in greater depth by both the students and practitioners. The first-year undergraduates particularly talked about the challenging nature of the concept:

> *How to organize it, reorder, number, or like link list, it took me longer to understand how to, like the more detailed stuff to do with arrays and different data structures. I found that more in-depth stuff harder than inheritance, encapsulation.*
>
> *(Year 1 student)*

The third-year undergraduates had an interesting discussion regarding the value of data structures, initially suggesting they weren't used at all. However, while doing the sorting activity there was some hesitation on how to position 'data structures' in their hierarchy of concepts. When prompted to explain their indecision, the participants began to discuss amongst themselves whether they fully understood the concept and recognized that it would have been worth the time to resolve this issue to improve project work in their final year:

> **Respondent 1** *(Data structures) weren't hard to understand, but we don't, we all understand, we should understand it a little bit better than what we currently do. And definitely the first year we felt that way.*
>
> **Respondent 2** *It was, I don't know, if we had put more effort for data structure for those interviews*[5] *I think it would have helped a lot.*
>
> **Respondent 3** *That's true.*

These discussions imply various forms of *partial understanding* associated with data structures. Initially this was found in the lack of recognition that data structures were a core concept. In the following discussion the idea emerged from the participants themselves that perhaps they had previously held a naïve version of knowledge in this area. They suggested that they were unable to apply the concept of data structures in different contexts. The practitioners' discussion revealed further insight as to how this partial understanding is perpetuated:

> *Incorrect data structures and algorithms: people generally don't do them correctly, so people just don't do them at all. Most of the stuff I see that's written avoids the use of having to use algorithms and data structures because they don't quite know how to do it and they try to re-use stuff that they can find that's been written by experts.*
>
> *(Practitioner)*

Despite the difficulties suggested by this participant, the practitioners still agreed that data structures were a fundamental aspect of programming, corroborating our suggestion that the undergraduate students had experienced a naïve version of knowledge:

> *This is the basics for any program writing, any good programmer should know data structures and algorithms.*
>
> *(Practitioner)*

We found strong evidence of partial understanding related to data structures. At the same time, both students and practitioners agreed that this was an essential concept. It is interesting to note, however, that professional programmers simply mention data structures as the "basics" without further discussion, while UG students consider them in much more detail and, as in the focus group quote above, appeared to realise that they may not have a sufficient understanding of this concept.

We can see two possible interpretations of these data. On the one hand, this may be an indication that data structures are an instance of a threshold concept tacitly known by professional programmers (and presumably teachers). On the other hand, this may also indicate that professional programmers themselves only have a naïve understanding of the concept (*cf.* also the first quote above). In day-to-day programming (outside of performance-critical or resource-constrained environments), elementary data structures like lists, trees, or maps will be reused from ready-made

[5]Referring to technical job interviews, which often focus on algorithmic and data-structure issues

libraries, often without any additional analysis. This may mean that, for many programming tasks, a superficial knowledge of data structure and a usage based on mimicry may be sufficient to get the job done. Possibly, professional developers have a tacit awareness that the implementation details of specific data structures do not matter for many contexts. While they acknowledge their importance generally, they are content to work with surface-level knowledge. Thus, these two interpretations may not necessarily be diametrically opposed.

4.1.4 Abstract classes. As programmers define classes and their features, they typically aim for a correspondence between the classes defined and objects that are meaningful in the domain for which the program is being developed. Sometimes, one identifies commonalities between a number of domain objects, but there are sufficiently many differences to justify creating separate classes for the domain objects. Abstract classes allow programmers to encapsulate the remaining commonalities in one class, while making it explicit that this class should not be instantiated directly.

The undergraduate students discussed the tricky nature of abstract classes, with some of them admitting that they still didn't understand them and another suggesting, as is illustrated in Table 2, that they didn't understand why they were taught, indicating a partial understanding of the concept. This latter perspective was shared by the third-year students, one of whom claimed, 'I've never used an abstract class'. The practitioners agreed that abstract classes were both troublesome and transformative, but did not discuss the concept in detail:

> **Interviewer** *Which did you really get stuck on, you know, possibly you found it very frustrating and it took a while for you to really get it?*
> **Respondent 1** *I think just this kind of group of abstract and interfaces* [points at some cards]*, particularly those concepts.*
> **Interviewer** [Reading the card labels] *Abstract classes, encapsulation and interfaces okay... What about in terms of, it doesn't necessarily have to be about feeling like a programmer, but it really was a big deal once you'd actually thought okay I get what this is now, this has made a big difference to how I understand programming?*
> **Respondent 2** *Well I guess it's all four of these.* [points at some cards]
> **Interviewer** *So these ones again* [Reading the card labels]*, so the abstract classes, encapsulation and interfaces, classes and inheritance.*

4.2 Additional Findings

As mentioned, to identify potential threshold concepts the data analysis was conducted in a largely deductive manner. However, the researchers maintained an open approach to data and as such made some interesting findings that were not threshold concepts in themselves, but could be related to our understanding of them. These findings are outlined below.

4.2.1 Threshold skills. As part of the lab observations one of the participants suggested that the nature of programming is skill-based and only makes sense in application. We therefore decided to consider the place of skills in the study. A similar notion has been put forward in some of the literature on threshold concepts [49]. One possible threshold skill was identified throughout the course of the study—*Code Organisation*—which was considered to be both troublesome and transformative by all of the focus groups:

> *Organisational code... is definitely one of the... it takes experience to write organized code.*
>
> *(Practitioner)*

> *I think you learn about code organisation throughout your life.*
>
> *(Year 3 student)*
>
> *You have to be organised not to lose any piece of code and to actually be able to find, like… adding, like, two lines of code.*
>
> *(Year 1 student)*

While there are a number of conceptual principles related to code organisation—for example, the general SOLID principles (*e.g.*, as described in [29])[6]—really mastering this topic requires gaining experience through application. Essentially, it is less about knowing the abstract concepts than about developing the skill of applying them in the right way in a given context.

A second potential threshold skill identified was *Designing Objects*. When developing object-oriented software, a key concern is how to define an appropriate set of objects and their inter-relations that can, together, solve the given problem effectively. This involves defining appropriate classes, relations between these classes, as well as methods implementing suitable collaborative behaviors between the objects that instantiate these classes. This skill is clearly related to Code Organisation.

Designing objects was singled out as *transformative* by some of the first-year students and *troublesome* or 'challenging' by others:

> *I do think objects would be quite challenging getting your head around… objects… which also goes hand in hand with the classes in inheritance thing.*
>
> *(Year 1 student)*

The third-year students agreed together that this skill was transformative and integrative:

> **Interviewer** *Anything else that you can think of that jumps out at you, something that once you got your head around it really changed your ideas of programming…?*
>
> **Respondent 1** *designing objects, it seems pretty simple now but once understand it it's…*
>
> **Respondent 2** *It's pretty good, knowing and understanding it.*
>
> **Respondent 1** *It helps your code organization, but it links into all these other things as well.*

We recognise it could be questioned whether the language used by the students indicates transformation. We suggest that in a group of (mainly male) peers it might be too much to ask undergraduates to describe powerful learning experiences. Instead, we are interpreting the language they are using as evidence of a type of learning experience that cannot be undone or unlearned, hence we consider it as evidence of the transformative aspect. Meanwhile, the practitioners described the skill as *troublesome* for them when they were learning to program.

4.2.2 Emotional response. Emotional responses frequently occurred or were referred to when a participant discussed a potential threshold concept, adding additional validity to claims of their legitimacy as such. Furthermore, it was perhaps unsurprising to find emotions appearing throughout the interview data. As has already been discussed, programming is difficult to learn and, it could

[6]The acronym stands for a list of principles for structuring code: The 'single responsibility principle' states that a class should only be responsible for one thing, the 'open / closed principle' is often described as requiring that "software entities should be open for extension, but closed for modification", 'Liskov's substitution principle' requires instances of classes to be usable wherever an instance of a super class was usable, the "interface segregation principle" asks for small-grained, role-specific interfaces, and the 'dependency inversion principle' asks that code should depend on abstract interfaces rather than concrete classes and let the specific classes be configured—for example through dependency injection.

be argued, difficult to practice. When a program is successful it can elicit a powerful emotional response:

> *I mean it's probably that if I have to do work I'd rather program because it's quite fun when you press run and something actually works. It pops up and you say oh my God* I've done something! *Like that's, I think that's one of the best things.*
>
> *(Year 1 student, emphasis added)*

As illustrated by the following quote from one of the practitioners, the challenges of programming continue well into a professional career, with the notion of success dependent on peer approval:

> *I've got ten years' professional experience, and I find it sometimes really intimidating when I go on [website] to look at comments, and someone will say you're not a good programmer, you don't know this or you don't know that*
>
> *(Practitioner)*

The above quotes point to the somewhat dramatic nature of learning to programme, whereby pushing one button enables the learner to see instant success or failure. This cliff-edge of getting the answer right or wrong is quite distinct from learning in other subject areas that require coping with more subjectivity. We believe the impact that this emotional journey has on the learner as they progress through threshold concepts is, as yet, under-explored. Given that our participants on the whole were more likely to talk about the emotions of learning to program, rather than the type of 'transformational' learning experiences commonly discussed in the threshold-concepts literature, it may be that the language of threshold concepts in programming needs to be nuanced. Eckerdal *et al.* seem to make a related point in their discussion of emotional response in [15].

4.2.3 Feeling like an 'insider'. Related to the emotions alluded to in the previous section was the complex notion of 'feeling like an insider'. Participants were asked about concepts which, upon mastering, made them feel 'like a programmer'. This was intended as a prompt to indicate transformation (as outlined in the theoretical framework, Sect 2.3). The responses of the practitioners in particular were quite surprising, as they did not convey confidence in feeling like an 'insider' despite having, in some instances, decades of professional experience:

> *A lot of people don't feel that confident in their ability, and so for me it doesn't feel like there was one concept that I grasped that necessarily made me feel like a programmer. It's more as you get more exposure to your colleagues and stuff, and I guess you're getting less criticism, you start to feel a little more confident*
>
> *(Practitioner 1)*

> *I don't know when I started thinking I'm a proper programmer but for me when my code goes through a code review successfully, a peer review successfully and sits in production without breaking…for say three, four weeks without any problem at all, then I start feeling proud, not proud, but content and satisfied with myself.*
>
> *(Practitioner 2)*

Both quotes once again reveal the range of emotional strings tied up with 'feeling like a programmer'. The second quote is a further suggestion of the drama involved in programming as a discipline, both to learn and to practice. Despite being an experienced software developer, the practitioner's satisfaction with his work came not with good design but when his software did not 'break'. The participant alludes to the peculiarity in programming of the work product being judged largely for its ability to not go wrong rather than because of inherently good design or execution. The students similarly referred to a sense of relief that a program simply functions as

required (see quote in Sect 4.2.2, above). The challenge is for learners to go beyond this stage to construct a program which is elegant and masterly.

4.2.4 Accidental complexities. An unexpected find in the data was the occurrence of what we termed 'accidental complexities'. Accidental complexities can be a concept that isn't a threshold concept, but which when taught alongside other larger (perhaps threshold) concepts, might introduce additional difficulty for the learners. These complexities are accidental because they are intended to enhance student learning, but may instead unintentionally end up getting in the way of learning. Examples of such concepts mentioned by student participants include Layout Managers—a specific concept in the context of graphical user interfaces—and anonymous inner classes, which were taught as a concise mechanism for implementing event handlers (part of the User Interface Architectures threshold concept). The latter seemed to trouble students partly because it introduces complicated syntax at a point when students are already struggling to make sense of a number of new concepts.

> *Like for example for the ActionListener, we did talk about it in the lectures at one point, didn't we? And he was talking about anonymous sub-classes* [sic] *and stuff like that, I sort of zoned out of that but anyway. But when you create an ActionListener you have a default, by default it makes like an ActionListener across the bottom so I knew that you had to put [unclear] in there but I didn't realise how to put it anonymous sub-class.*
>
> *(Year 1 student)*[7]

Accidental complexities may also occur when attempting to provide the students with an additional skill (*e.g.,* teamwork), or when using a pedagogical device (*e.g.,* pair programming as a means of enabling peer teaching). While the inclusion of such activities would generally be seen to be good practice, several of the student participants discussed the problematic nature of trying to amalgamate different people's ideas together to find a cohesive solution to the problem at hand. This was cited as a particular issue when students were in teams of mixed experience or ability:

> *The major course work it's a lot harder than the minor course work was, because obviously it's a more complex problem but at the same time there's more people. So having more people is itself a problem because you're going to try and get more people's ideas into one cohesive [whole].*
>
> *(Year 1 student)*

5 DISCUSSION

This study had the opportunity to use a a number of different lenses to explore conceptual difficulties in programming education (*cf.* Sect. 2). We chose a framework based on threshold-concept theory, as the idea of uncovering the conceptual "jewels" in programming [41, 46, 47] was particularly fitting for our aim of improving teaching of programming. However, as a result of this study, we are less convinced about the viability of a threshold-concept lens in achieving these aims. In particular, as other researchers before us, we faced challenges in clearly identifying concepts as troublesome or transformative, and finding the right granularity of concepts to identify. In this section, we discuss these issues in detail, followed by a discussion of the limitations of the research undertaken and implications for pedagogy and research.

[7]Note that students associate ActionListener with the concept of event handlers as this is the primary context in which they have been taught about event handlers.

5.1 The challenges of defining threshold concepts

Using the dimensions of *troublesome* and *transformative* as part of the analysis framework should make the initial identification of threshold concepts relatively straightforward. However, identifying concepts with these properties in the focus-group data was more challenging than we originally expected:

(1) *Challenges of identifying* troublesome *concepts.* In many cases, troublesome concepts were directly identified by participants. However, in our second analysis round, we found evidence of over-confidence where students did not report a concept as troublesome, but their subsequent discussions revealed that they had only a partial understanding of the concept. It seems clear that explicitly identifying concepts as troublesome can be difficult for participants, as also found by Shinners-Kennedy and Fincher [44]. An important reason for this seems to be that troublesome concepts often lead to *partial understanding*, which prevents participants from actively identifying their troublesome nature. We argue that sustained partial understanding of a concept can be a strong indicator that someone is in liminal space; that is, that the concept is indeed sufficiently troublesome to be a candidate threshold concept. Conversely, however, once participants have passed through the liminal space, it becomes difficult for them to remember that the concept was ever difficult. The concept has become tacit knowledge, which may also make it more difficult to teach explicitly to others.

(2) *Challenges of identifying* transformative *concepts.* There is some evidence of the transformative effect of certain concepts in our data—for example, the emotional response some participants conveyed when reporting mastering a task associated with a particular concept. However, it could be challenged whether these transformations are really as substantial as the definition of threshold concepts suggests:

> "A threshold concept can be considered as akin to a portal, opening up a new and previously inaccessible way of thinking about something. It represents a transformed way of understanding, or interpreting, or viewing something without which the learner cannot progress." [33]

However, we argue that for any given discipline there are likely only very few concepts that have such a substantial transformative effect. While the transformation they achieve undoubtedly has a substantial impact on a student's ability to grasp a field, identifying other troublesome concepts with a smaller transformative effect is at least as important for the design of effective pedagogies.

Identifying *troublesome* concepts—while a more complex process than may have been previously thought—should be possible with the considerations indicated above. However, we are cautious of the extent to which the concepts identified through our study fulfil the profoundly *transformative* nature seemingly expected within threshold concept literature. We argue that there can be gradients of transformation, and within programming education particularly we suggest that a 'soft' transformation occurs for students when they master the concepts proposed within this paper. We feel that identifying concepts associated with such transformation is of use from a pragmatic, pedagogical perspective (see Sect. 5.5). Nevertheless, we recognise that the argument for 'soft' transformation is somewhat contingent on one's perspective regarding identifying a few broad threshold concepts *vs.* many precise ones. This issue is discussed further in the next section.

5.2 'Broadness' or 'granularity' of threshold concepts?

As mentioned at the beginning of this paper, there has been discussion in the literature regarding the suitability of object orientation as a threshold concept candidate, with arguments that it

covers too broad an area. This study attempted to provide some granularity and suggested 'classes and inheritance' and 'abstract classes' as potential candidates. These findings correspond with those identified by other studies. For example, Sanders and McCartney [42] suggest inheritance, polymorphism, object interaction and software objects as candidates for threshold concepts in OO. Sien et al. [45] suggested classes and objects were part of a family of associated threshold concepts situated in OO. Meanwhile, the "Sweden Group" of researchers have suggested that objects and classes are merely difficult sub-concepts of OO, rather than threshold concepts in themselves [13]. We strongly feel that OO is too big to be a useful candidate threshold concept, especially from a perspective on pedagogy. Instead, the concepts identified through our study (as well as possibly those identified in [42, 45]) are of smaller granularity but still sufficient difficulty and provide enough transformation in students' understanding to be more useful as a basis for pedagogic interventions.

We suggest that the lack of consensus on identified threshold concepts in programming indicates that attempts to pinpoint only one or two 'jewels' of the discipline are not viable, or may not be appropriate, in this field. We propose that it is suitable to investigate a larger number of key but precise concepts, be they sub-concepts of broad concepts such as OO, or narrow stand-alone concepts such as 'user interface architectures'. The challenge associated with such an approach is determining how many concepts is *too many*, in order to avoid the problem of a 'stuffed' curriculum [9].

5.3 The bumpy road to becoming an 'expert' programmer

The suggestion of acquiring a new identity as part of a professional community has been considered a significant aspect of threshold concept theory in the literature, as it is indicative of transformation and subjectivity. Nevertheless, we suggest the notion of 'feeling like an insider' may be limited in its capacity to help identify potential threshold concepts in this subject area. The students themselves did not respond to prompting regarding 'feeling like a programmer' and many of the industry practitioners joked they still didn't feel like they were a programmer, even after years of professional experience. Instead, when discussing the notion the practitioners made several references to seeking validation from their community, suggesting that 'feeling like an insider' has more meaning as a social threshold—as found in Wegerif's work [50]—than a conceptual one. Social rather than conceptual learning in programming may be an area that deserves further study.

The notion of 'feeling like a programmer' is very close to the idea of 'communities of practice' [27]. We know from empirical research in software engineering (*e.g.,* [8, 39]) that ideas of apprenticeship and legitimate peripheral participation play a role in the workplace learning of professional developers. In the more formally structured context of most university-level computer-science courses, these ideas of social learning seem to play a lesser role. It would be interesting to explore ways in which legitimate peripheral participation can be better integrated into university teaching of computer science concepts (*e.g.,* through more systematic application of problem-based learning [12, 17, 24]) and how such a change in teaching would affect students developing a self-perception as 'programmers'. Research on newcomer socialisation [3] and learner identity [38] is beginning to address some of these questions.

5.4 Limitations

We believe our study provides valuable new insights into teaching and learning programming, and the sampling of practitioners in particular is seen as a significant contribution of this work. However, we recognize there are some limitations to the generalizability of our findings. Some of these were unavoidable from a methodological perspective—for example the participants were

self-selecting and the students were sampled from one institution with a mixed ability cohort. Other possible limitations were because of deliberate decisions on the research design. For example, all data collection was undertaken by the first author, a non-expert in programming. This could have affected the direction of the discussions taking place in the focus groups, but it was felt this was necessary to ensure participants could freely discuss concepts they found troublesome without apprehension. Focus groups have been criticised for their ability to produce a 'group effect', where dominant voices suppress a potentially legitimate minority view and for being logistically difficult to organise [6]. The practical limitations of organising focus groups did mean that we sampled fewer participants than we would have liked, as it was difficult to arrange for a convenient time for several people to meet and sometimes participants turned up late, or did not turn up at all. Nevertheless, we believe focus groups to be the most appropriate method to obtain data that was representative of the cohort being sampled and related to the more nuanced aspects of the subject matter. Furthermore, the process of group interaction resulted in some participants critically reflecting upon assumptions they made regarding concepts they found (or didn't find) difficult.

A further example of a potential limitation was the use of the concept organization activity as a prompt for participants, based on concepts covered in the course curriculum. To mitigate the bias this may have caused, the activity was done after an open discussion with participants where they were free to consider any concept they found troublesome or transformative. Furthermore, they were encouraged to include any concepts they considered missing from those offered in the activity itself. The use of curriculum-based prompts also allowed for discussion by participants as to why they thought the concept was taught and allowed us to analyze whether they had fully or partially understood the concept.

An important challenge for the study was the lack of systematic baseline measure of participants' programming experience. While the differentiation between first year and third year undergraduates is useful, the lack of such baseline data on a sample so diverse in its nature (as discussed in the introduction) means we must be cautious in generalizing conclusions about threshold concepts to other students in the same or similar cohorts. This is further compounded by the fact that some of the concepts identified, usually the 'straightforward' threshold concepts, were a result of consensus within the focus group, while others arose from one individual's suggestion within a focus group. Nevertheless, comparison of the concepts identified within the first-year undergraduate, the third-year undergraduate and the practitioner focus groups allowed us to find the commonality between these stakeholders and therefore we argue the concepts offered above can hold a legitimate claim to be troublesome concepts in programming education, with 'soft' transformative aspects (see Sect. 5.1). Furthermore, the 'collective' approach largely mitigated any hindsight bias from the third-year students and the practitioners, while comparison between the students and practitioners provided an opportunity to explore which naïve versions of knowledge students may have at different stages in their education.

Some of the concepts identified could themselves be considered rather large at this point. This is particularly true for 'data structures', which is almost as big as the—often criticised—'object orientation'. More research will be needed to break up this concept into smaller components (*cf.* Sect. 5.2).

A final limitation is methodological: while we have collected data from a wider range of participants (1st-year students through practitioners) than typically considered in the existing literature, we have only interacted with each participant once and have only collected self-reported perceptions of difficulty. This makes it difficult to extract information about actual difficulty of a given concept. We have partially mitigated this concern by using focus groups, which have allowed participants to discuss their assessment of (relative) difficulty of concepts, so that we can report on

a community consensus of concept difficulty. However, a better understanding of concept difficulty and the development of understanding (including the degree of 'transformativeness' of a concept) can only be achieved using a different methodology—for example, a longitudinal approach using activity-based data collection (*e.g.,* think-aloud tasks or concept maps [36]). We hope to be able to undertake such a study in the future.

5.5 Implications for pedagogy and threshold concept research

The findings produced by this study have built on existing research to identify suitable threshold concepts within programming education. Suggestions for further research have been threaded through the discussion section of this paper in response to the limitations of the study, the findings themselves and the theory with which they were discerned. Some additional implications have been generated as a result of these discussions, these are discussed in more detail here.

Accidental complexities were an interesting finding in the focus-group data, contributing towards existing debates regarding tensions between accepted good teaching practice and threshold concepts and skills [1, 26]. The complexities may have been introduced because of an attempt to provide students with additional tools and skills which, while in themselves undeniably useful (*e.g.,* learning to work as a team or using anonymous inner classes when writing event handlers), may actually cause them difficulties in grasping the main concept at hand. For instance, anonymous inner classes were considered troublesome by students, who associated them only with event handlers, part of the identified threshold concept User Interface Architecture. Anonymous inner classes were taught only in this context. Given that students find both ideas challenging, we recommend that they should not be taught at the same time. In the most recent iteration of the module, we have removed explicit teaching of anonymous inner classes, limiting the idea to a "stretch concept" targeting more advanced students.

In cases where smaller concepts, identified as contributing towards accidental complexity, are considered essential knowledge for a proficient programmer, there is a strong argument to delay teaching them until the main threshold concept has been mastered. Further exploration of the satellite concepts taught alongside identified threshold concepts may reveal additional opportunities to strip back and simplify the curriculum, using the 'less is more' approach advocated in other TC literature [9, 26, 46, 47]. This will allow students to spend more time on the concepts which take priority and ensure they have successfully traversed their 'liminal space'. As Sorva [47] points out, a student who has crossed a threshold is better placed to learn new, related concepts more easily.

Additionally, it seems worth reconsidering which points in the curriculum can be taught as group work: group work creates its own challenges, combining these with learning a threshold concept may be a step too far. While there may be strong justification for using a new pedagogical approach at the same time as introducing a threshold concept, it is argued that the changes this approach brings or the scale of 'complexity' it introduces (*e.g.,* team size, level of independence of the teams, etc.) can negatively impact on students passing through liminal space.

We suggest that interventions responding to the identification of threshold concepts should consider the nature of liminal space associated with them. Threshold concepts that are straightforwardly troublesome and transformative have a potentially more observable period of liminality, thus the amount of additional time given for the acquisition of the concept and removal of accidental complexities should likewise be more straightforward. Conversely, threshold concepts associated with partial understanding, where the liminal space is more dynamic, may be more difficult to factor in. Nevertheless, any intervention should factor additional time to embed the importance of the threshold concept in question, for students to have experience applying threshold concepts in a variety of different contexts, and an opportunity to return to them at a later stage to challenge any

naïve versions of knowledge. A longitudinal study tracing students' evolving understanding of threshold concepts and correlating this with the teaching they have been exposed to would be very helpful in answering many of these questions. Indeed, our methodological choice of gathering data from three cohorts at different points in their journey to 'becoming a programmer' was strongly motivated by a desire to capture at least some of this temporal development, which is not usually done in the existing literature.

The characteristic of "feeling like an insider" is interesting in the context of programming: even the professional programmers we talked to did not consider themselves "expert programmers". In this study, we did not undertake a comparative analysis of the language used by participants at different stages in talking about individual concepts. Such changes in language are considered an element of the "feeling like an insider" characteristic [34] and would be very interesting to study in future research.

Based on our findings and the wider literature on threshold concepts in programming [42], interesting questions for research remain. In particular, how can the identification of threshold concepts specifically inform the curriculum and approaches to teaching programming? Does the order in which concepts are taught make a difference to how challenging they are? How can we best decide what auxiliary concepts and skills are essential to be taught alongside threshold concepts and which are just unhelpful noise? What is the role of threshold skills in programming education? As ever, more research is needed to explore these areas.

6 CONCLUSIONS

This study has expanded on existing research on threshold concepts in programming education by sampling professional software developers as well as undergraduate students, the former representing an as-yet un-investigated population in threshold concept research. In addition to this methodological contribution, the study proposed four candidates for threshold concepts, identified through a systematic process of discerning recurring concepts between different stakeholders and comparing against our theoretical framework. While many of the concepts discussed in the focus groups were identified as troublesome, only the four put forward above were also considered to be transformative by one or more of the participant groups. As a result, there is compelling evidence to suggest that those concepts are suitable candidates for threshold concepts in programming. Of particular interest are *Classes and Inheritance* and *Abstract Classes,* which fall under the area of Object-Oriented programming (OOP). OOP had previously been identified [41] in the literature as an area which was too large to be of any significant use as a threshold concept; our findings contribute more specific sub-concepts of OOP which may be of more help in identifying points in the curriculum where students may require additional support.

Beyond these concepts, we also contribute to the wider discussion of the suitability of the threshold-concept lens in programming education. The difficulties we and others have faced in clearly identifying a set of threshold concepts lead us to ask whether, in particular, the *transformative* aspect of the threshold-concept definition is asking too much in the context of programming and will, invariably, lead to the identification of concepts that are too broad and not useful for the continued improvement of pedagogical approaches. We have found some indication of 'soft' transformation in our data and argue that this is a good basis for identifying important concepts to be the focus of a curriculum. However, whether that transformation is big enough to make these concepts 'threshold' remains up for debate.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their thorough reading and thoughtful comments, which helped substantially improve this paper.

REFERENCES

[1] Caroline Baillie, John A Bowden, and Jan H F Meyer. 2012. Threshold capabilities: threshold concepts and knowledge capability linked through variation theory. *Higher Education* 65, 2 (June 2012), 227–246.
[2] S. Barradell. 2012. The identification of threshold concepts: a review of theoretical complexities and methodological challenges. *Higher Education* 65, 2 (June 2012), 265–276.
[3] Andrew Begel and Beth Simon. 2008. Novice software developers, all over again. In *Proc. 4th Int'l Workshop on Computing Education Research (ICER'08)*. 3–14. DOI: http://dx.doi.org/10.1145/1404520.1404522
[4] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. 2007. Threshold concepts in computer science: Do they exist and are they useful? *ACM SIGCSE Bulletin* 39, 1 (March 2007), 504.
[5] Neil C. C. Brown, Sue Sentance, Tom Crick, and Simon Humphreys. 2014. Restart: The Resurgence of Computer Science in UK Schools. *ACM Trans. Comput. Educ.* 14, 2, Article 9 (June 2014), 22 pages. DOI: http://dx.doi.org/10.1145/2602484
[6] Alan Bryman. 2012. *Social Research Methods* (4th edition ed.). Oxford University Press.
[7] Michael Clancy. 2004. Misconceptions and Attitudes that Interfere with Learning to Program. In *Computer Science Education Research*, Sally A Fincher and Marian Petre (Eds.). Taylor & Francis, London, 85–100.
[8] Alistair Cockburn and Laurie Williams. 2001. The costs and benefits of pair programming. *Extreme programming examined* (2001), 223–248.
[9] Glynis Cousin. 2006. An introduction to threshold concepts. *Planet* 17, 1 (Dec. 2006), 4–5.
[10] P. Davies. 2003. Threshold concepts: How can we recognise them? (2003). Embedding threshold concepts project: working paper 1.
[11] Martyn Denscombe. 2012. *The Good Research Guide: For Small-Scale Social Research Projects* (4th edition ed.). McGraw-Hill Education (UK).
[12] J. Doody. 2009. *A Longitudinal Evaluation of the Impact of a Problem-Based Learning Approach to the Teaching of Software Development in Higher Education.* Ph.D. Dissertation. University of Durham.
[13] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. 2006. Putting Threshold Concepts into Context in Computer Science Education. In *Proc 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)*. ACM, 103–107. DOI: http://dx.doi.org/10.1145/1140124.1140154
[14] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. 2006. Categorizing student software designs: Methods, results, and implications. *Computer science education* 16, 3 (Feb. 2006), 197–209.
[15] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, Lynda Thomas, and Carol Zander. 2007. From Limen to Lumen: Computing Students in Liminal Spaces. In *Proceedings of the Third International Workshop on Computing Education Research (ICER '07)*. ACM, New York, NY, USA, 123–132. DOI: http://dx.doi.org/10.1145/1288580.1288597
[16] Steve Furber. 2012. *Shut down or restart? The way forward for computing in UK schools.* Technical Report. The Royal Society.
[17] M. García-Famoso. 2005. Problem-based learning : a case study in computer science. In *International Conference on Multimedia and ICT in Education*. Portugal, 1–5.
[18] Anabela Gomes and António José Mendes. 2007. An Environment to Improve Programming Education. In *Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech '07)*. ACM, New York, NY, USA, Article 88, 6 pages. DOI: http://dx.doi.org/10.1145/1330598.1330691
[19] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding Object Misconceptions. In *Proc 28th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'97)*. ACM, 131–134. DOI: http://dx.doi.org/10.1145/268084.268132
[20] Janet Hughes and D. Ramanee Peiris. 2006. ASSISTing CS1 Students to Learn: Learning Approaches and Object-oriented Programming. *SIGCSE Bull.* 38, 3 (June 2006), 275–279. DOI: http://dx.doi.org/10.1145/1140123.1140197
[21] Tony Jenkins and John Davy. 2002. Diversity and Motivation in Introductory Programming. *Innovation in Teaching and Learning in Information and Computer Sciences* 1, 1 (2002), 1–9. DOI: http://dx.doi.org/10.11120/ital.2002.01010003
[22] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proc. 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, 107–111. DOI: http://dx.doi.org/10.1145/1734263.1734299
[23] G Kamberelis and G Dimitriadis. 2005. Focus groups: Strategic articulations of pedagogy, politics, and inquiry. In *The SAGE Handbook of Qualitative Research* (3rd edition ed.), Norman K Denzin and Yvonna S Lincoln (Eds.). Sage

Publications, 887–907.

[24] Judy Kay, Michael Barg, Alan Fekete, Tony Greening, Owen Hollands, Jeffrey H Kingston, and Kate Crawford. 2000. Problem-Based Learning for Foundation Computer Science Courses. *Computer Science Education* 10, 2 (2000), 109–128. DOI: http://dx.doi.org/10.1076/0899-3408(200008)10

[25] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. *SIGCSE Bull.* 37, 3 (June 2005), 14–18. DOI: http://dx.doi.org/10.1145/1151954.1067453

[26] Ray Land, Glynis Cousin, Jan H F Meyer, and Peter Davies. 2012. Implications of threshold concepts for course design and evaluation. In *Overcoming barriers to student understanding: threshold concepts and troublesome knowledge*, Jan H F Meyer and Ray Land (Eds.). Routledge.

[27] Jean Lave and Etienne Wenger. 1991. *Situated Learning: Legitimate peripheral participation.* Cambridge University Press.

[28] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bull.* 36, 4 (June 2004), 119–150. DOI: http://dx.doi.org/10.1145/1041624.1041673

[29] Robert C. Martin. 2013. *Agile Software Development, Principles, Patterns, and Practices.* Pearson.

[30] Richard E. Mayer. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (March 1981), 121–141. DOI: http://dx.doi.org/10.1145/356835.356841

[31] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '01).* ACM, New York, NY, USA, 125–180. DOI: http://dx.doi.org/10.1145/572133.572137

[32] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St. Clair, and Lynda Thomas. 2006. A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. *SIGCSE Bull.* 38, 4 (June 2006), 182–194. DOI: http://dx.doi.org/10.1145/1189136.1189185

[33] Jan Meyer and Ray Land. 2003. Threshold concepts and troublesome knowledge: linkages to ways of thinking and practising. In *Improving Student Learning – Theory and Practice Ten Years On*, C. Rust (Ed.). Oxford Centre for Staff and Learning Development (OCSLD), 412–424.

[34] Jan H. F. Meyer and Ray Land. 2005. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education* 49, 3 (2005), 373–388. DOI: http://dx.doi.org/10.1007/s10734-004-6779-5

[35] Jan Erik Moström, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, and Carol Zander. 2008. Concrete Examples of Abstraction As Manifested in Students' Transformative Experiences. In *Proc. 4th Int'l Workshop on Computing Education Research (ICER '08).* ACM, New York, NY, USA, 125–136. DOI: http://dx.doi.org/10.1145/1404520.1404533

[36] Andreas Michael Mühling. 2014. *Investigating Knowledge Structures in Computer Science Education.* PhD thesis. Technische Universität München, Germany.

[37] M. Pedroni, B. Meyer, and M. Oriol. 2009. *What do beginning CS majors know?* Technical Report 631. ETH Zürich, Chair of Software Engineering.

[38] Anne-Kathrin Peters. 2018. Students' Experience of Participation in a Discipline: A Longitudinal Study of Computer Science and IT Engineering Students. *ACM Trans. Comput. Educ.* 19, 1 (2018), 5:1–5:28. DOI: http://dx.doi.org/10.1145/3230011

[39] Laura Plonka, Helen Sharp, Janet Van der Linden, and Yvonne Dittrich. 2015. Knowledge transfer in pair programming: An in-depth analysis. *International Journal of Human Computer Studies* 73 (2015), 66–78. DOI: http://dx.doi.org/10.1016/j.ijhcs.2014.09.001

[40] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (2003), 137–172. DOI: http://dx.doi.org/10.1076/csed.13.2.137.14200

[41] Janet Rountree and Nathan Rountree. 2009. Issues Regarding Threshold Concepts in Computer Science. In *Proc. 11th Australasian Conference on Computing Education - Volume 95 (ACE '09).* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 139–146. http://dl.acm.org/citation.cfm?id=1862712.1862733

[42] Kate Sanders and Robert McCartney. 2016. Threshold Concepts in Computing: Past, Present, and Future. In *Proc. 16th Koli Calling Int'l Conference on Computing Education Research (Koli Calling '16).* ACM, New York, NY, USA, 91–100. DOI: http://dx.doi.org/10.1145/2999541.2999546

[43] Dermot Shinners-Kennedy and Sally Fincher. 2015. Scaffolded autoethnography: a method for examining practice-to-research. In *6th Research in Engineering Education Symposium.* http://kar.kent.ac.uk/51057/

[44] Dermot Shinners-Kennedy and Sally A. Fincher. 2013. Identifying Threshold Concepts: From Dead End to a New Direction. In *Proc. 9th Annual Int'l ACM Conf. on International Computing Education Research (ICER '13)*. ACM, 9–18. DOI: http://dx.doi.org/10.1145/2493394.2493396

[45] Ven Yu Sien and David Weng Kwai Chong. 2011. Threshold Concepts in Object-Oriented Modelling. In *7th Educators' Symposium, MODELS 2011: Software Modeling in Education*. 1–11.

[46] Juha Sorva. 2010. Reflections on threshold concepts in computer programming and beyond. In *10th Koli Calling International Conference*. ACM Press, 21–30.

[47] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2 (June 2013), 1–31.

[48] The Royal Society. 2017. After the Reboot: computing education in UK schools. Royal Society report. (Nov. 2017). https://royalsociety.org/~/media/policy/projects/computing-education/computing-education-report.pdf

[49] Lynda Thomas, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan-Erik Moström, Kate Sanders, and Carol Zander. 2014. A broader threshold: Including skills as well as concepts in computing education. In *National Academy's 6th Annual Conf. and the 4th Biennial Threshold Concepts Conf. (NAIRTL'14)*. 154–158.

[50] Rupert Wegerif. 1998. The Social dimension of asynchronous learning networks. *Journal of Asynchronous Learning Networks* 2 (1998), 34–49.

[51] Carol Zander, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Kate Sanders. 2008. Threshold Concepts in Computer Science: a multinational empirical investigation. In *Threshold Concepts within the Disciplines*. Sense Publishers, Rotterdam, 105–118. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-15763

A FOCUS GROUP PROMPTS

The following is an indicative list of prompts used during the focus groups. The focus groups with professional developers were undertaken after those with students and we adjusted the questions to provide linkage between both parts of the study. We also used different kinds of questions with professional developers to accommodate for differences in context and expected level of experience.

- Student focus groups:
  (1) Could you give me an idea of the sort of experience you have had with programming before starting the course?
  (2) Which are your favourite parts of programming?
  (3) Which are your least favourite aspects?
  (4) Which aspects of programming would you say you would like more help with?
  (5) What would you change about the course with regard to programming?
  (6) How good would you say you are at programming?
- Professional developer groups:
  (1) Can you give a brief explanation of your experience of programming, including whether you learned it at university or otherwise?
  (2) Any concepts that you got particularly stuck on?
  (3) Did you ever use concepts where you simply copied what you had seen, only later understanding them properly (or perhaps not at all?)
  (4) What were the ideas or concepts in programming that, once you had grasped them, made you think and feel as a 'proper' programmer?
  (5) In the focus groups, students identified (insert appropriate potential threshold concept) as a particularly challenging/useful topic area. What are your thoughts on this?
  (6) Which aspects of object-orientation (insert other appropriate potential threshold concept) would you say are particularly troublesome to grasp/provided a significant shift in your understanding of programming?

B CARD TOPICS PROVIDED

The following is a list, in alphabetical order, of concepts used in the sorting exercise component of the focus groups. These were based on the programming curriculum for first year undergraduates at King's:

- Abstract Classes
- Arrays (Data Structures)
- Assignments
- Class Diagram Notation
- Classes and Inheritance
- Code Organisation
- Conditional Statements
- Console Output
- Debugging
- Designing Objects
- Encapsulation
- Exceptions
- Getting from a Problem to a Program
- IDEs
- Interfaces
- Java Concepts for GUI Programming
- Layout Management Concepts
- Lists (Data Structures)
- Loop Statements
- Modelling Real-World Entities
- Packages
- Processing Numeric Data
- Reading Input Strings
- Regular Expressions
- User Interface Architectures
- Variables, References, Objects
- Version Control

C OCCURRENCES OF CONCEPTS IN THE DATA

In this appendix, we quantitatively summarise our data on candidate concepts based on the coding of all focus group transcripts as well as of photographs taken of the results of the card-sorting activity.

Table 3 summarises the occurrence counts of data coded for any potential candidate concepts throughout all our data, separated into occurrences during the free discussion and occurrences during the sorting activity. Note that this is occurrence data only, any specific occurrence may or may not be related to the concept being troublesome or transformative. It can be seen that a number of concepts came up during the free discussion already, before participants received more direct prompts from the card-sorting activity.

Table 4 shows the concepts that were introduced by participants during the card-sorting activity. These are concepts that were not on one of the cards provided, but where participants felt the need to add a new card with the concept on it. Overall, this was a rare occurrence. Note that only some of these concepts also show up in the transcript annotations, because not all of them were actually talked about by the participants.

Table 3. Occurrence counts of potential candidate concepts coded throughout the data. For each focus group, the 'Free' column indicates the number of occurrences of codes for the concept before the card-sorting activity, while the 'Sorting' column indicates the number of occurrences during the card-sorting activity.

	Practitioners		UG Y1 Group 1		UG Y1 Group 2		UG Y1 Group 3		UG Y3		Total		
Concept	Free	Sorting	Free	Sorting	Free	Sorting	Free	Sorting	Free	Sorting	Free	Sorting	Overall
Abstract Classes	0	2	0	0	0	0	0	2	0	1	0	5	5
Abstraction	4	0	0	0	0	0	0	0	0	0	4	0	4
Algorithms and 'big O' notation	0	4	0	0	0	0	0	0	0	0	0	4	4
Class Diagram Notation	0	0	0	0	0	0	0	2	0	1	0	3	3
Classes and inheritance	1	1	0	3	0	3	0	1	0	1	1	9	10
Concurrency	0	5	0	0	0	0	0	0	0	0	0	5	5
Data structures	4	0	0	1	0	0	0	1	0	3	4	5	9
Designing objects	0	1	0	2	0	0	0	1	0	1	0	5	5
Encapsulation	0	2	0	0	0	0	0	0	0	0	0	2	2
Exceptions	0	0	0	1	0	0	0	1	0	1	0	3	3
IDEs	0	1	0	0	0	0	0	2	0	1	0	4	4
Interfaces	0	2	0	0	0	3	0	3	0	1	0	9	9
Java concepts for GUI programming	0	0	1	0	0	0	0	2	0	0	1	2	3
Layout Management Concepts	0	1	0	2	0	0	0	0	0	0	0	3	3
Modelling Real World Entities	0	0	0	0	0	1	0	0	0	0	0	1	1
Moving from a problem to a solution	2	0	2	0	0	0	0	1	0	0	4	1	5
Object-oriented programming	1	2	2	0	0	1	0	0	0	0	3	3	6
Packages	0	2	0	0	0	1	0	1	0	1	0	5	5
Polymorphism	0	1	0	2	0	0	0	0	0	2	0	5	5
Recursion	3	5	0	0	0	0	0	0	0	0	3	5	8
Regular Expressions	0	3	0	1	0	1	0	0	0	0	0	5	5
User Interface Architectures	0	1	0	1	0	1	0	5	0	0	0	8	8
Variables, References and Objects	0	0	0	0	0	0	0	1	0	0	0	1	1
Version Control	1	2	0	1	0	3	0	2	0	3	1	11	12

Table 4. Concepts introduced during the card-sorting activity. A 1 in a column indicates that the participants added a new card with this concept during their card-sorting activity

Concept	Practitioners	UG Y1 Group 1	UG Y1 Group 2	UG Y1 Group 3	UG Y3	Total
Polymorphism	1	1	0	0	0	2
Reading and writing files	0	0	0	1	0	1
External APIs	0	0	0	1	0	1
Algorithms and 'big O' notation	1	0	0	0	0	1
Concurrency	1	0	0	0	0	1
Functional programming	1	0	0	0	0	1
Recursion	1	0	0	0	0	1